Polymorphism 多型

Polymorphism (多型) is an important mechanism in OOP. We treat the base-class and its many derived-classes equally. As a result, we have very flexible code.

General and common properties and operations (of different derived-classes) appear in the base-class. Specific and specialized operations are often **overridden** in the derived-class.

The Basics

Let us consider a base-class `Shape` with derived-classes `Circle` and `Triangle`:

Shape.h

```
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream>

class Shape {
public:
    Shape() { std::cout << "Shape::Shape()" << std::endl; }
    virtual void draw() const { std::cout << "Shape::draw()" <<
                                              std::endl; }
};

class Circle : public Shape {
public:
    Circle() { std::cout << "Circle::Circle()" << std::endl; }
    void draw() const { std::cout << "Circle::draw()" <<
                                              std::endl; }
};

class Triangle : public Shape {
public:
    Triangle() { std::cout << "Triangle::Triangle()" <<
                                              std::endl; }
    void draw() const { std::cout << "Triangle::draw()" <<
                                              std::endl; }
};
#endif
```

UseShape.cpp

```
#include "Shape.h"

void display(const Shape& s){
    s.draw();
}

int main(){
    Shape* s = new Shape();
```

```
    display(*s);
    Shape* c = new Circle();
    display(*c);
    Shape* t = new Triangle();
    display(*t);

    delete s;
    delete c;
    delete t;
    return 0;
}
```

```
Shape::Shape()
Shape::draw()
Shape::Shape()
Circle::Circle()
Circle::draw()
Shape::Shape()
Triangle::Triangle()
Triangle::draw()
```

The method `display` takes a parameter of the `Shape` type. We can call `display` by passing any object of `Shape` and its derived-class (`Circle`, `Triangle`).

This is commonly known as polymorphism (from a Greek word meaning "many forms"). **Polymorphism means an object of a derived-class can be used wherever its base-class object is used.** 香蕉 跟 芭樂 可以在任何 水果 可以被使用的情況，被使用。

Q: Where is the flexibility?
A:
With polymorphism, we can write flexible pieces of code that **do not need to change** when we add new derived-classes into the program.

15.4.2 Serious Polymorphism: Abstract Base-Class (ABC)

For cleaner design, we often like to define the base-class as an **abstract base-class (ABC).** ABCs are pure interfaces. Functions can be declared in the base-class without being implemented. This type of functions is called **pure virtual functions**[1]. Derived-classes will then **need to** define the function with their own implementation.

A class is an ABC if it has at least one pure virtual function. We cannot make objects of ABCs.

ABCs are useful when implementation of functions that cannot be provided in the base-class,

[1]Pure Virtual Functions and Abstract Classes in C++ https://www.geeksforgeeks.org/pure-virtual-functions-and-abstract-classes/

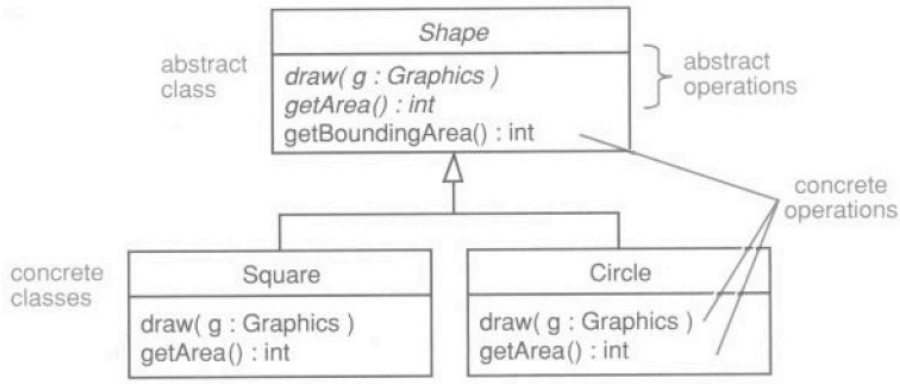
because we do not know the implementation. For example, let `Shape` be a base-class. Without knowing the exact shape, we cannot provide the implementation of the function `draw()` in `Shape`. Derived-classes (triangle, circle) have its own implementation of `draw()`.

A pure virtual function is specified by writing = **0** after the function parameter list.

A pure virtual function provides an interface to be overridden but **cannot be called in this class**. (We cannot even instantiate an ABC object.)

```
class Test { // An abstract base-class
public:
    virtual void show() = 0; // Pure Virtual Function
};
```

The *Shape* class is an ABC. Its definition looks like:



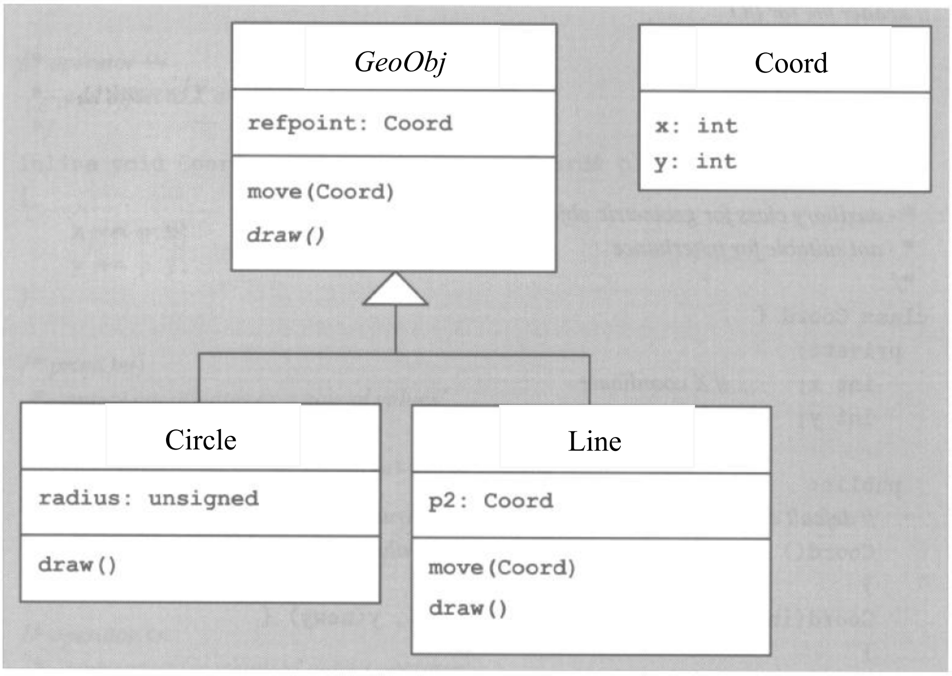
Note the italic font in the pure virtual functions in `Shape`.

```
class Shape {
public:
    virtual void draw(Graphics g) = 0;
    virtual int getArea() = 0;
    int getBoundingArea(); // to be implemented in Shape.cpp
    virtual ~Shape() = default;
}
```

Remarks:

1. An **abstract method** is implemented through a pure virtual function. It means this member function must be **overridden**.
2. A class with **abstract methods** is an ABC.
3. Any object pointed by a pointer of the ABC type is **an object of a derived-class** of that **ABC**. The derived-class is required to implement <u>all</u> pure virtual functions.

<u>Example</u>: Let us now work through an example to demonstrate polymorphism through the ABC design. A `GeoObj` class hierarchy shown below.



Before we dive into abstract base-class, let's take a quick look of the auxiliary class `Coord`

Coord.h

```
#ifndef COORD_H
#define COORD_H
#include <iostream>

class Coord {
  private:
    int x;     // X coordinate
    int y;     // Y coordinate
  public:
    // default constructor, and two-parameter constructor
    Coord() : x(0), y(0) {}    // default values: 0
    Coord(int newx, int newy) : x(newx), y(newy) {}

    Coord operator + (const Coord&) const;    // addition
    Coord operator - () const;               // negation
    void  operator += (const Coord&);         // +=
    void  printOn(std::ostream& strm) const;  // output
};

inline Coord Coord::operator + (const Coord& p) const {
    return Coord(x+p.x,y+p.y);
}

inline Coord Coord::operator - () const {
    return Coord(-x,-y);
}

inline void Coord::operator += (const Coord& p) {
    x += p.x;
    y += p.y;
}
```

```
inline void Coord::printOn(std::ostream& strm) const {
    strm << '(' << x << ',' << y << ')';
}

inline std::ostream& operator<< (std::ostream& strm, const Coord&
p){
    p.printOn(strm);
    return strm;
}

#endif // COORD_H
```

In this example, two types of geometric objects `Line` and `Circle` are regarded as being geometric objects under the common base-class `GeoObj` (the ABC).

All geometric objects have a reference point. All geometric objects can be moved with `move()` and drawn with `draw()`. For `move()`, there is a default implementation that simply moves the reference point accordingly.

A `Circle` additionally has a radius and implements the `draw()` function (is it necessary?). The function for moving is inherited (what does it mean?).

A `Line` has a second point (the first point is the reference point) and implements the `draw()` function. `Line` also re-implements the function for moving (what does it mean?).

The ABC `GeoObj` defines the **common** attributes and operations:

GeoObj.h

```
#ifndef GEOOBJ_H
#define GEOOBJ_H

#include "Coord.h"

class GeoObj {
  protected:
    // every GeoObj has a reference point
    Coord refpoint;
    GeoObj(const Coord& p) : refpoint(p) {}

  public:
    virtual void move(const Coord& offset) {
        refpoint += offset;
    }
    virtual void draw() const = 0;
    virtual ~GeoObj() = default;
};

#endif  // GEOOBJ_H
```

Circle.h

The derived-class `Circle`

```
#ifndef CIRCLE_H
#define CIRCLE_H

// header file for I/O
#include <iostream>

#include "GeoObj.h"

class Circle : public GeoObj {
  protected:
    unsigned radius;    // radius

  public:
    // constructor for center point and radius
    Circle(const Coord& m, unsigned r) : GeoObj(m), radius(r) {}

    // draw geometric object (now implemented)
    virtual void draw() const;

    // virtual destructor
    virtual ~Circle() {}
};

inline void Circle::draw() const {
    std::cout << "Circle around center point " << refpoint
              << " with radius " << radius << std::endl;
}

#endif // CIRCLE_H
```

Line.h

The derived-class `Line`

```
#ifndef LINE_H
#define LINE_H

#include <iostream>
#include "GeoObj.h"

class Line : public GeoObj {
  protected:
    Coord p2;    // second point, end point

  public:
    Line(const Coord& a, const Coord& b): GeoObj(a), p2(b) {}

    virtual void draw() const;
    virtual void move(const Coord&);
    virtual ~Line() {}
};

inline void Line::draw() const {
    std::cout << "Line from " << refpoint
```

```
              << " to " << p2 << std::endl;
}

inline void Line::move(const Coord& offset) {
    refpoint += offset;    // represents GeoObj::move(offset);
    p2 += offset;
}

#endif // LINE_H
```

<u>UseGeoObj.cpp</u>

Application example

```
#include "Line.h"
#include "Circle.h"
#include "GeoObj.h"

// forward declaration
void printGeoObj(const GeoObj&);

int main() {
    Line l1(Coord(1,2), Coord(3,4));
    Line l2(Coord(7,7), Coord(0,0));
    Circle c(Coord(3,3), 11);

    // array as an inhomogenous collection of geometric objects:
    GeoObj* coll[3];

    coll[0] = &l1;     // collection contains: - line l1
    coll[1] = &c;      //     - circle c
    coll[2] = &l2;     //     - line l2

    /* move and draw elements in the collection
     * - the correct function is called automatically
     */
    for (int i=0; i<3; i++) {
        coll[i]->draw();
        coll[i]->move(Coord(3,-3));
    }

    // output individual objects via auxiliary function
    printGeoObj(l1);
    printGeoObj(c);
    printGeoObj(l2);
}

void printGeoObj(const GeoObj& obj){
    obj.draw();
}
```

```
Line from (1,2) to (3,4)
Circle around center point (3,3) with radius 11
Line from (7,7) to (0,0)
Line from (4,-1) to (6,1)
Circle around center point (6,0) with radius 11
Line from (10,4) to (3,-3)
```

Another case to illustrate the advantages of polymorphism: a derived-class capable of combining multiple geometric objects together to form a group of geometric objects.

GeoGroup.h
```
#ifndef GEOGROUP_H
#define GEOGROUP_H

#include <vector>
#include "GeoObj.h"

class GeoGroup : public GeoObj {
  protected:
    std::vector<GeoObj*> elems;

  public:
    GeoGroup(const Coord& p = Coord(0,0)) : GeoObj(p) {}

    virtual void draw() const;

    virtual void add(GeoObj&);

    virtual bool remove(GeoObj&);

    virtual ~GeoGroup() = default;
};

#endif  // GEOGROUP_H
```

GeoGroup.cpp
```
#include "GeoGroup.h"
#include <algorithm>

void GeoGroup::add(GeoObj& obj) {
    // keep address of the passed geometric object
    elems.push_back(&obj);
}

void GeoGroup::draw() const {
   for (const auto& e : elems) {
       e->move(refpoint);   // add offset for the reference point
       e->draw();          // draw element
       e->move(-refpoint);  // subtract offset
   }
}

bool GeoGroup::remove(GeoObj& obj) {
    // find first element with this address and remove it
    // return whether an object was found and removed
    std::vector<GeoObj*>::iterator pos;
    pos = std::find(elems.begin(),elems.end(),&obj);
    if (pos != elems.end()) {
        elems.erase(pos);
        return true;
    }
```

```
    else {
        return false;
    }
}
```

UseGeoGroup.cpp

Application example with `GeoGroup` derived-class

```
#include <iostream>

#include "Line.h"
#include "Circle.h"
#include "GeoGroup.h"

int main() {
    Line l1(Coord(1, 2), Coord(3, 4));
    Line l2(Coord(7, 7), Coord(0, 0));
    Circle c(Coord(3, 3), 11);

    GeoGroup g;

    g.add(l1);           // GeoGroup contains: - line l1
    g.add(c);            //     - circle c
    g.add(l2);           //     - line l2

    g.draw();            // draw GeoGroup
    std::cout << std::endl;

    g.move(Coord(3, -3));  // move offset of GeoGroup
    g.draw();                // draw GeoGroup again
    std::cout << std::endl;

    g.remove(l1);         // GeoGroup now only contains c and l2
    g.draw();            // draw GeoGroup again
}
```

```
Line from (1,2) to (3,4)
Circle around center point (3,3) with radius 11
Line from (7,7) to (0,0)

Line from (4,-1) to (6,1)
Circle around center point (6,0) with radius 11
Line from (10,4) to (3,-3)

Circle around center point (6,0) with radius 11
Line from (10,4) to (3,-3)
```

Remarks: What's the beauty of introducing `GeoGroup`?

1. Note that interface of the `GeoGroup` hides the internal use of pointers. Thus, the application programmers need only pass the objects that need to get inserted or removed.
2. The `GeoGroup` contains no code that refers to **any concrete type** of the geometric objects it contains. Thus, if a new geometric object, such as `Triangle` is introduced, we only need to make sure that `Triangle` is derived from `GeoObj` and that's it.

Another example with container using dynamic memory

```cpp
#include <iostream>
#include <vector>

class Animal {// abstract base class
public:
    virtual void speak() = 0; // pure virtual method
    virtual ~Animal() {  std::cout << "~Animal()" << std::endl;  }
};

class Dog : public Animal {// derived class
public:
    // polymorphic implementation of speak
    virtual void speak() { std::cout << "Ruff!" << std::endl; }
    virtual ~Dog() {  std::cout << "~Dog()" << std::endl;  }
};

class Cat : public Animal {// derived class
public:
    // polymorphic implementation of speak
    virtual void speak() { std::cout << "Meow!" << std::endl; }
    virtual ~Cat() {  std::cout << "~Cat()" << std::endl;  }
};

using namespace std;

int main( int argc, char* args[] ){
    // container of base class pointers
    vector<Animal*> barn;

    // dynamically allocate Animal instances for the container
    barn.push_back( new Dog() );
    barn.push_back( new Cat() );

    // invoke the speak method for elements in the container
    for( auto i = barn.begin(); i != barn.end(); ++i )
      (*i)->speak(); // dereference iterator to get Animal* object

    // free the allocated memory
    for(auto &a : barn)
      delete a;

    return 0;
}
```

```
Ruff!
Meow!
~Dog()
~Animal()
~Cat()
~Animal()
```